# Assignment 2
# CS-351
# Spring 2015
**Due Date: 4/25/2015 at 11:59 pm (No extensions)**

## Objectives:

1. To demonstrate IPC principles in action.
2. To develop greater appreciation of shared memory, message passing, and signal IPC mechanisms.
3. To implement a multiprocess application.
4. To use message passing, shared memory, and signals IPC mechanisms in order to implement cooperation between between processes.
5. To develop a practical application where the sender process transfers files to the receiver process.

## Overview

In this assignment you will use your knowledge of shared memory and message passing in order to implement a multiprocess application which synchronously transfers files between two processes.

You shall implement two related programs: a *sender program* and a *receiver program* as described below:

- **The Sender Program:** this program shall implement the process that sends files to the receiver process. The sender shall be invoked as `./sender <FILE NAME>` where `sender` is the name of the executable and `<FILE NAME>` is the name of the file to transfer. For example, `./sender song.mp3`. When invoked, the sender shall perform the following sequence of steps:
    1. Attach to the shared memory segment and connect to the message queue both previously set up by the receiver process.
    2. Send the name of the file specified at the command line to the receiver process using the message queue. The message shall contain a field called `fileName` specifying the name of the file to to be sent.
    3. Read a predefined number of bytes from the specified file, and store these bytes in the shared memory segment.
    4. Send a message to the receiver (using the message queue). The message shall contain a field called `size` indicating how many bytes were read from the file and are currently stored in the shared memory segment.
    5. Wait on the message queue to receive a message from the receiver confirming successful retrieval of data from shared memory and saving of data to the file.

6. Go back to step 3. Repeat until the whole file has been transferred.
7. When the end of the file is reached, send a message to the receiver with the `size` field set to 0. This will tell the receiver that the sender has nothing more to send.
8. Close the file, detach the shared memory segment, print the number of bytes sent, and exit.

- **The Receiver Program:** this program shall implement the process that receives files from the sender process. The program shall be invoked as `./recv` where `recv` is the name of the executable. When invoked, the receiver program shall perform the following sequence of steps:

  1. Setup a shared memory segment and a message queue.
  2. Wait on the message queue to receive a message containing the name of the file about to be transferred. The received message shall contain a field called `fileName` specifying the name of the file to be transferred.
  3. The program shall wait on the message queue to receive a message from the sender program. The received message shall contain a field called `size` specifying the number of bytes the sender has saved in the shared memory segment that are ready to be read by the receiver.
  4. If `size` is not 0, then read `size` number of bytes from shared memory and save them to the file with the same name as the file received in step 2 with `_recv` appended to the end of the file name. For example, if the sender sent a file named `song.mp3`, the receiver saves the received file as `song.mp3_recv`. Finally, send a message to the sender acknowledging successful reception and saving of data and go back to step 3.
  5. Otherwise, if `size` is 0, then close the file, deallocate the shared memory segment, deallocate the message queue, print the number of bytes received, and exit.

- When user presses `Ctrl-C` in order to terminate the receiver, the receiver shall deallocate the shared memory and the message queue and then exit. This can be implemented by setting up a signal handler for the `SIGINT` signal. Code illustrating how to do this has been provided in (`signaldemo.cpp`).

## Technical Details

The skeleton codes for sender and receiver can be found on Titanium. The files are as follows:

- `sender.cpp:` the skeleton code for the sender (see the TODO: comments in order to find out what to fill in)
- `recv.cpp:` the skeleton code for the receiver (see the TODO: comments in order to find out what to fill in).
- `msg.h:` the header file used by both the sender and the receiver. It contains two structs representing the different messages sent via the message queue. The `fileNameMsg` struct represents the message used by the sender to send the name of the file to be transferred to the receiver:
  - `long mtype:` specifies the message type.

  - `char fileName[100]`: specifies the name of the file.

- The `message` struct represents the message used by the sender to inform the receiver of the number of bytes in the shared memory segment that are ready to be received. It features the following fields:
  - `long mtype`: represents the message type.
  - `int size`: the number of bytes written to the shared memory.
- The `ackMessage` struct represents a message sent by the receiver to the sender acknowledging the successful reception and saving of data to the file. It comprises one field, `long mtype`, which specifies the type of the message.
- In addition to the structures, `msg.h` also defines macros representing three different message types.
  - `SENDER_DATA_TYPE`: macro representing the type of message sent from the sender to the receiver informing the receiver of the number of bytes stored in the shared memory that are ready to be received. The value of this macro is integer 1. The `mtype` field of all messages of type `message` should always be set to this macro.
  - `RECV_DONE_TYPE`: macro representing the type of message sent from receiver to the sender acknowledging successful reception and saving of data. The value of this macro is integer 2. The `mtype` field of all messages of type `ackMessage` should always be set to equal to this macro.
  - `FILE_NAME_TRANSFER_TYPE`: macro representing the type of message containing the name of the file. The value of this macro is 3. The `mtype` field of all messages of type `fileNameMsg` should always be set equal to this macro.
  - `MAX_FILE_NAME_SIZE`: macro representing the maximum length of the file name in the fileNameMsg struct.
- `Makefile`: enables you to build both the sender and the receiver by simply typing "make" at the command line.
- `signaldemo.cpp`: a program illustrating how to install a signal handler for the `SIGINT` signal. The `SIGINT` signal is sent to the process when user presses Control-C.

**Please note: by default the sender and the receiver skeleton programs will give you errors when you run them. This is because they are accessing unallocated, unattached regions of shared memory. It's your job to fill in the appropriate functionality in the skeleton, denoted by the TODO comments in order to make the programs work**.

The following links provide additional documentation about shared memory and message queues:

- Message Queues: http://beej.us/guide/bgipc/output/html/multipage/mq.html
- Shared Memory: http://beej.us/guide/bgipc/output/html/multipage/shm.html

## BONUS

Implement ***separate*** versions of sender and receiver which instead of relying on message queues to signal events, rely purely on signals. Hint: use handlers for `SIGUSR1` and `SIGUSR2`.. Hint: how can you tell the receiver the number of bytes in the shared memory?

## SUBMISSION GUIDELINES:

- ***This assignment MUST be completed using C or C++ on Linux.***
- You may work in groups of 3.
- ***Your assignment must compile and run on the TITAN server.*** Please contact the CS office to for an account.
- Please hand in your source code electronically (do not submit .o or executable code) through **TITANIUM**.
- You must make sure that the code compiles and runs correctly.
- Write a README file (text file, do not submit a .doc file) which contains
  - Your name and email address.
  - The programming language you used (i.e. C or C++).
  - How to execute your program.
  - Whether you implemented the extra credit.
  - Anything special about your submission that we should take note of.
- Place all your files under one directory with a unique name (such as `p2-[userid]` for assignment 1, e.g. `p2-mgofman1`).
- Tar the contents of this directory using the following command. `tar cvf [directory_name].tar [directory_name]` E.g. `tar -cvf p2-mgofman1.tar p2-mgofman1/`
- Use TITANIUM to upload the tared file you created above.

## Grading guideline:

- Program compiles: 5'
- Correct use of message queues: 30'
- Correct use of shared memory: 30'
- Sender and receiver correctly output the number of bytes sent and received 5'
- Program deallocates shared memory and message queues after exiting: 10'.
- Correct file transfer: 10'
- Correct signal handling: 5'

- All system calls are error-checked: 5'
- README file: 5'
- Bonus: 10'
- Late submissions shall be penalized 10%. No assignments shall be accepted after 24 hours.

## Academic Honesty:

**Academic Honesty:** All forms of cheating shall be treated with utmost seriousness. You may discuss the problems with other students, however, you must write your **OWN codes and solutions**. Discussing solutions to the problem is **NOT** acceptable (unless specified otherwise). Copying an assignment from another student or allowing another student to copy your work **may lead to an automatic F for this course**. Moss shall be used to detect plagiarism in programming assignments. If you have any questions about whether an act of collaboration may be treated as academic dishonesty, please consult the instructor before you collaborate. Details posted at http://www.fullerton.edu/senate/documents/PDF/300/UPS300-021.pdf.